Topology-aware Camera Control for Real-time Applications

Alberto Jovane, Amaury Louarn, Marc Christie

Topology-aware Camera Control for Real-time Applications

Alberto Jovane
alberto.jovane@inria.fr
Inria, CNRS, IRISA
Rennes, France

Amaury Louarn
amaury.louarn@irisa.fr
Univ Rennes, Inria, CNRS, IRISA, M2S
Rennes, France

Marc Christie
marc.christie@irisa.fr
Univ Rennes, Inria, CNRS, IRISA, M2S
Rennes, France

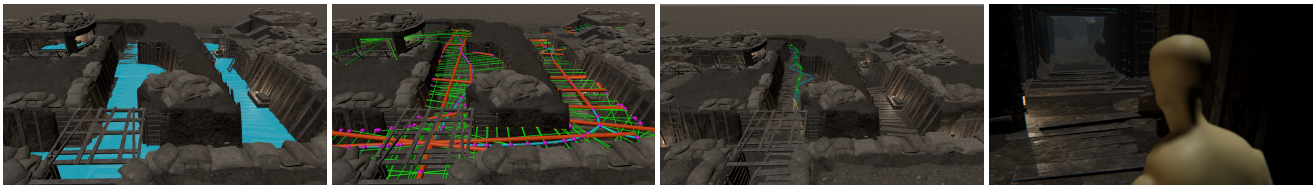


Figure 1: Our topology-aware camera control system works as follows: starting from a virtual environment with its navigation mesh in blue (left), a collection of camera tracks are generated by clustering points obtained via ray casts (green) generated from a topological skeleton representation of the navigation mesh (center-left).The camera is then controlled in real-time by a physical system that follows a target on the best camera track in order to film an actor navigating in the environment (center-right) and the final result (right).

ABSTRACT

Placing and moving virtual cameras in real-time 3D environments is a task that remains complex due to the many requirements which need to be satisfied simultaneously. Beyond the essential features of ensuring visibility and frame composition for one or multiple targets, an ideal camera system should provide designers with tools to create variations in camera placement and motions, and create shots which conform to aesthetic recommendations. In this paper, we propose a controllable process that will assist developers and artists in placing cinematographic cameras and camera paths throughout complex virtual environments, a task that was often manually performed until now. With no specification and no previous knowledge on the events, our tool exploits a topological analysis of the environment to capture the potential movements of the agents, highlight linearities and create an abstract skeletal representation of the environment. This representation is then exploited to automatically generate potentially relevant camera positions and trajectories organized in a graph representation with visibility information. At run-time, the system can then efficiently select appropriate cameras and trajectories according to artistic recommendations. We demonstrate the features of the proposed system with realistic game-like environments, highlighting the capacity to analyze a complex environment, generate relevant camera positions and camera tracks, and run efficiently with a range of different camera behaviours.

CCS CONCEPTS

• **Computing methodologies** → **Procedural animation**; • **Applied computing** → **Media arts**.

KEYWORDS

Virtual Cinematography, Camera Behaviors

, ,

Alberto Jovane, Amaury Louarn, and Marc Christie. 2020. Topology-aware Camera Control for Real-time Applications. In *Proceedings of* . , 9 pages.

1 INTRODUCTION

With the increasing quality in 3D models and animations, coupled with real-time realistic rendering capacities, 3D computer games are proposing immersive experiences ever closer to real cinematographic experiences. A key component of player immersion is how the camera is placed and moved according to scene contents, players motions as well as narrative and stylistic constraints. This requires the ability to decide at run time, while the user is playing, the best camera angles and displacements which satisfy such constraints in complex 3D environments.

To date, most cinematographic camera systems rely either on (i) the prior manual placement of cameras by artists in 3D environments which are then triggered at run-time by events in the game (*e.g.* characters entering a building, climbing stairs, jumping between platforms), or (ii) the use of motion planning techniques through the computation of camera roadmaps in the 3D environment (*e.g.* probabilistic roadmaps [Li and Cheng 2008; Nieuwenhuisen and Overmars 2004]) which are exploited at runtime but generally fail in creating a cinematographic look-and feel. Even for games in which cameras are directly controlled by the players, there are pressing requirements to create well-shot cinematographic sequences from single or multiple playing sessions that could then be streamed to larger audiences, typically for e-sports games. Based on prior work, some solutions have been proposed to control the selection of cameras and cuts between cameras but the provision of a general and automated system able to populate a given 3D environment with cinematographic camera angles and camera paths to track characters in a cinematographic way remains unaddressed.

Creating such a system requires to address the following challenges (i) automatically create camera angles and camera tracks of cinematographic quality (ii) connecting camera angles and camera tracks together in a joint representation to enable continuous or discrete transitions and (iii) at run-time, computing the camera motion

and cuts given a number of targets to follow and high-level constraints (static *vs.* dynamic cameras, shot sizes, anticipation *vs.* lazy cameras, cutting pace). Addressing the problem first requires a better understanding of underlying motivations and constraints which guide the design of camera in real movies and endow them with a cinematographic look-and-feel. A first observation is that this design is predominantly a matter of directorial style. For the same motion of characters, there are significant variations in how the cameras can be placed and moved [Jiang et al. 2020]. Therefore a artistic control over the camera parameters is required.

A second observation is that camera tracks are strongly driven by the topology of the environment. For example, when considering the design of a camera sequence in a corridor, there is little number of alternatives in trajectories: motions all follow the shape of the corridor, generally in a close to linear motion where possible, tracking characters from front, side or rear view. In addition, linear or close to linear camera motions are prevalent in real movies, first due to physical constraints of camera rigs (mostly linear rails or camera dolly carts), and second due to their visual simplicity (complex motions tend to distract the spectator from the content, unless it is the intention).

At last, static cameras are commonplace in movies. When tracking characters, these cameras are placed at locations which maximise visibility, and generally pan to follow characters motions (unless implementing specific intentions such as cameras placed behind hedges to enforce partial visibility).

We noted the following requirements:

- populating the environment with static cameras observing large areas.
- populating the environment with linear camera motions that simulate classical dolly track motions.
- populating the environment with a network of linked camera paths which would enable following a character without cuts whatever the motion they performs.

To address these requirements, we propose a *topology-aware* approach designed in two phases. A first offline phase that exploits navigation meshes in 3D gaming environments to build a simplified skeletal representation. Omni-directional or controlled directional ray-casts are then performed from the skeletal representation to the scene geometry, to populate the environment with virtual cameras along the scene geometry and aiming at the skeleton. Virtual cameras are then clustered using sequential RANSAC with a linear model to extract pieces of linear camera motions. Finally, all linearized motions are linked in a graph representation. A second and online phase that compute at each frame a virtual target position on the edges of this graph, representing an optimal camera position and then a physical camera model is used to attract the virtual camera towards the optimal camera.

Our contributions are threefold:

- a novel approach to automatically compute a collection of camera angles and camera tracks which are aware of the scene topology and implement different directorial styles, using a sampling+clustering approach;
- a graph representation dedicated to camera control: the *camera navigation graph* which abstracts the regions in which the camera can move, enables efficient queries, and yields smooth camera motions;
- a real-time cinematographic system which can compute in real-time (in less than 20ms), smooth camera motions and automated transitions between viewpoints, responding to high-level directorial constraints (camera distance, camera angle, cutting speed, static or dynamic tracking)

As a result, this opens many possibilities for real-time fully automated cinematographic systems deployed in game engines with complex environments and interactive control of directorial style, such as in e-sports live casting events, where game sessions can be conveyed in more cinematographic ways by strongly borrowing and adapting techniques from real movies.

2 RELATED WORK

Camera control deals with issues in placing and moving cameras in virtual environments [Christie et al. 2008]. It is a well-studied problem in computer graphics, and approaches have been exploring how visual features such as target visibility, screen composition, optimal view or camera smoothness can be enforced [Christie and Olivier 2006] by relying on motion-planning, optimization and more recently deep-learning techniques [Jiang et al. 2020]. In the context of real-time 3D applications such as game engines, contributions have essentially focused on target tracking techniques [Halper et al. 2001] and coupling visibility with path-planning techniques [Oskam et al. 2009].

2.1 Automated generation of camera paths

The computation of camera paths with a prior knowledge of the environment is either performed as a path planning process or motion planning process (*i.e.* integrating temporal information). Different planning techniques have been proposed to guide the motions of cameras based on the underlying representations proposed in the robotics literature (see [Lino et al. 2010; Oskam et al. 2009]). For example in [Oskam et al. 2009], the authors relied on a prior spherical decomposition of the free-obstacle space, by filling the space with intersecting spheres. Visibility between each pair of spheres is also precomputed using ray-casting and stored. A graph-based roadmap of the environment can then be constructed where each node is the center of a sphere, and each edge is a collision-free motion from one sphere to neighbor intersecting one. At run-time the roadmap is queried with an A* algorithm to compute the shorted path from the current camera position to the target position that maximises the visibility of a target. To highlight the motion of a vehicule, Huang [Huang et al. 2016] rely on an interactive optimisation technique which computes a sequence of waypoints that will ensure the proper tracking of targets by the camera. Key characteristics to optimize are visibility of the target, camera smoothness, and visual load (the more objects in the scene, the slower the camera is).

The key issue common to most path or motion planning approaches is actually how to characterize what makes a good cinematographic motion. While smoothness (expressed as the absence of jerks on the evolution of camera trajectories) is often considered, there is no clear consensus on characteristics of good cinematographic motions. In [Galvane et al. 2015a], the authors propose

to create camera trajectories by performing interpolations of cinematographic properties in the screen space (angle on targets, composition of targets on the screen, size of targets). The idea is also exploited by Galvane *etal.* in [Galvane et al. 2018] to create camera paths for drone motions that would avoid sudden on-screen changes.

Most approaches however only focus on the computation of a single camera, or camera path, to perform the requested task, and have not been addressing the issue of populating environments with cinematographic cameras.

2.2 Maximum coverture issue

As far as we know there is little literature available on the automated placing of cinematographic cameras in 3D environments, driven by the topology of the environment.

Some previous work address the issue of automated camera placement typically in the context of the Art Gallery problem [O'Rourke 1987]. This is a well-known optimization problem where the goal is to place the minimum number of surveillance cameras to cover the entire surface of an art gallery. One example is [van den Hengel et al. 2009] where they used a genetic algorithm to place the cameras given a 3d model.

Other approaches, such as Chittaro etal [Chittaro et al. 2010], proposed the design of an authoring tool that generates virtual tours to ease the navigation process, yet the specification of POIs (points of interest) are defined manually. On our side, we are not interested in the minimum number of cameras, nor a limited set of POIs but to obtain qualitative views on some possible events inside the scene, which are not known beforehand. The particular challenge we face here is the computation of camera locations and motions without knowing beforehand the motion of characters and events.

2.3 Automated cinematography techniques

Automated placement of multiple cameras has also been addressed in the specific case of designing staging and shooting layouts. In [Louarn et al. 2018], the authors rely on a high-level specification language to place both the camera and the characters in relation with the environment. The work deals with the optimization of events visualization, but the main difference is that here the positioning of the cameras is in relation with the positioning of the agents, and both tasks are addressed at the same time. In our work we have no information of where the agents will be, so we have to rely on hypotheses as to where the characters will be and how they will move, and ensure that there are enough cameras to cover their range of positions/motions.

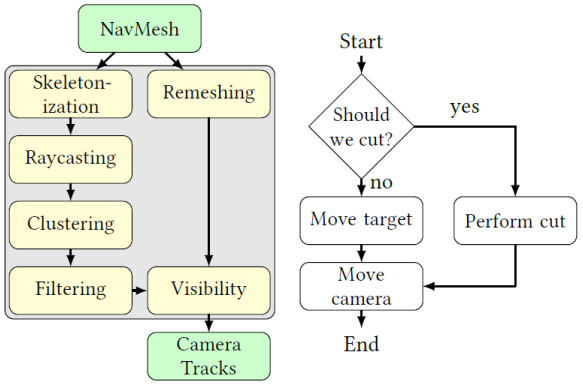
3 OVERVIEW

Our approach provide a real time generation of cinematic cameras in game-like environment, through two stages: The offline pre-processing stage (detailed in Section 4) takes as input a *navigation map* – a 3D topology which represents the surface on which characters can navigate in 3D environments. A geometric skeleton is extracted from the topology to provide an abstract and simplified representation of the navigation map. The skeleton is then used as a baseline on which (i) a raycast sampling is performed, by shooting rays locally orthogonal to the skeleton towards the 3D environment. Hits of the rays and samples from the skeleton compose a collection of camera poses. Then with a sequential RANSAC process we perform a multi-model estimation, where our model is linear pieces of camera motions. Linear motions are further cleaned, and structured into a camera navigation graph.

The second process, detailed in Section 5, uses the camera navigation graph to decide in real-time where to place and how to move the camera according to the position of an entity. Designers can tune some elements such as framing and cutting strategies to influence the camera placement in real-time.

4 PRECOMPUTATION

The first stage of our system consists in an off-line computation, the input of which is a navigation mesh (navmesh) a 3D triangulated polygon, subset of the environment. A *navmesh* is a common representation used in 3D applications for navigation agents, which encodes the surfaces on which the agents can move. Navmeshes are supported by all mainstream 3D game engines (such as Unity and Unreal Engine). As displayed in Fig. 2 (a), this process is separated in six distinct steps, which are later described.



(a) Off-line process **(b)** On-line process

Figure 2: Overview of the two stages of our system.

Skeletonization. A skeleton [Aichholzer et al. 1996; Brandt and Algazi 1992] of the navigation mesh is extracted, it provides an abstraction of the topological characteristics of an arbitrary environment (corridors, intersections, forks, dead-ends…).

Raycast sampling. A sampling process using raycasts from the skeleton to the 3D environment along heuristic directions then creates a cloud of possible camera positions either along the environment (if the rays hit the environment) or in mid-air (to a cut-off distance if there is no hit).

Clustering. A clustering of the possible cameras is performed using a multi-model fitting algorithm (here a sequential RANSAC [Fischler and Bolles 1981] for it's $O(n)$ performance) to extract a collection of underlying linear sections which will become camera tracks.

Filtering. A filtering stage is performed to remove specific artifacts from the clustering (eg a track that collides with the environment).

Dual visibility estimation. To reduce the cost of visibility computation at run-time, we estimate the visibility between camera nodes and triangles from the navigation mesh, in a way similar to Oskam *et al.* [Oskam et al. 2009] using Monte-Carlo raycast sampling. To increase precision in the estimation, a mesh refinement is performed on the navigation mesh to obtain triangles under a given area [Botsch and Kobbelt 2004]. Visibility estimation is stored both in the camera nodes (the list of triangles visible from this camera) and in the triangles (the list of cameras which see this triangle).

Building a camera navigation graph. The last stage finally links the isolated cameras and camera tracks into a *camera navigation graph* which can be efficiently queried to decide where to place and how to move the camera.

The output of this process is (i) a camera navigation graph representing possible camera locations (the nodes) and possible camera tracks (the edges),and (ii) the visibility information relative to a remeshed navigation surface.

4.1 Skeletonization

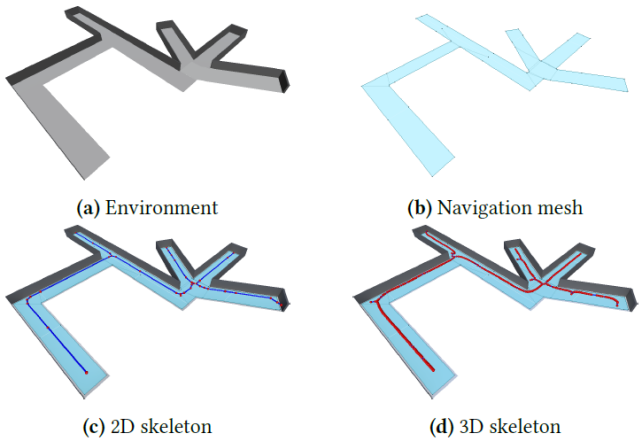


Figure 3: 2D and 3D skeletonization of a navigation mesh without overlaps in height results in different skeleton representations.

The purpose of this first stage is to obtain a simplified and abstract representation of where entities (*e.g.* characters) can navigate in a given 3D environment. The first step of the process is to extract a topological structure of the environment. We propose to rely on the navigation mesh which is an approximation of the environment that can be automatically computed [Lamarche 2009; Oliva and Pelechano 2011; Xiang Xu 2011] and obviously offers a complete representation of where entities can be located. This information remains however complex to process and analyse if corridors, intersections or forks need to be detected. To both abstract and simplify this representation, we propose a topological skeleton extraction from the navigation mesh using straight skeletons [Aichholzer et al. 1996] and mean curvature skeletons [Tagliasacchi et al. 2012].

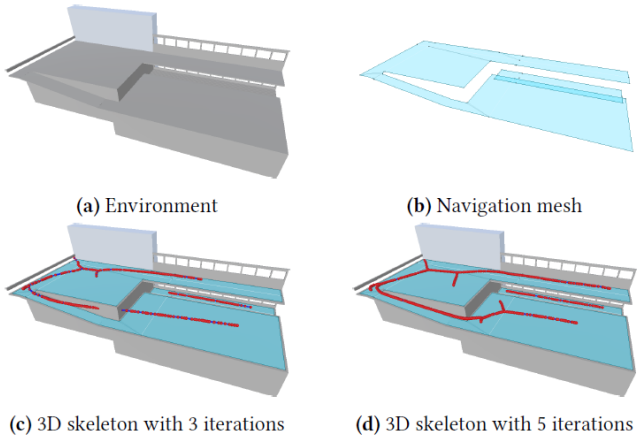


Figure 4: 3D skeletonization of a navigation mesh with overlaps in height. Notice the influence in the number of edge-split iterations on the resulting skeleton: with 3 iterations (c) it intersects the environment and with 5 iterations there are no intersections (d).

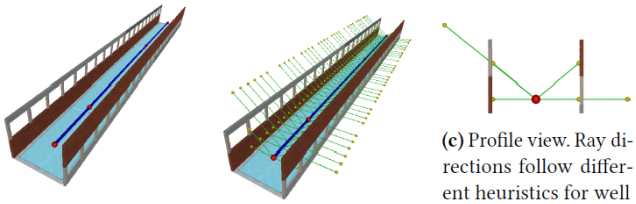
Straight skeletons were introduced by Aichholzer *et al.* as a replacement of widely used medial axis techniques, for its lower computational cost and simple straight-line structure. A straight skeleton is solely made of line segments which are pieces of angular bisectors of polygon edges, and computed using a shrinking process on possibly non convex polygons. Straight skeletons are limited to 2D polygons only. Therefore, for all navigation meshes where projection on a 2D plane does not yield overlapping surfaces, we simply (i) perform the straight skeleton extraction on the 2D projected navigation mesh and (ii) reproject the skeleton vertices to the original navigation mesh.

For navigation meshes where 2D projection overlap, we propose to rely on mean curvature skeletons [Tagliasacchi et al. 2012]. The mean curvature technique collapses a given 3D mesh into a skeleton structure using mean curvature flow and Voronoi medial skeleton to obtain a medially centered curve skeleton. Well centered curve skeleton is computed by minimizing the energy function E:

$$E = E_{\text{smooth}} + E_{\text{velocity}} + E_{\text{medial}}$$

where E_{medial} energy pulls the evolving surface towards the medial axis, at an energy velocity E_{velocity} depending on the curvature, with a smoothness controlled by energy E_{smooth}. To apply this technique we (i) first extrude the navigation mesh by a height representing the size of an entity (typically the character) navigating on this mesh, (ii) then perform edge-split iterations to refine the mesh (as described in [Tagliasacchi et al. 2012] to improve quality) and (iii) apply the mean curvature technique.

In terms of computational cost, the straight skeleton technique is more efficient (*e.g.* 0.3s for 3 (a) versus 9.7s for 3 (b). Also, while the quality of the 3D skeleton gets better with a more complex input mesh, the computational time gets higher (*e.g.* 0.4s for 3 edge-split iterations in 4 (c) versus 7.7s for 5 iterations in 4 (d)).



(a) Test environment, navmesh (in cyan) and skeleton (in blue).

(b) Casted rays (green) and resulting points.

(c) Profile view. Ray directions follow different heuristics for well placed cameras (here at level or 45° angle).

Figure 5: Raycast sampling on a simple environment with walls and windows. The skeleton (red and blue), the rays (green), and the resulting points (yellow). Note how some rays intersect the environment while others go through the windows/open areas.

4.2 Raycast sampling

The skeletal representation provided in the previous stage abstracts the motion of the characters on the navigation mesh to a sequence of segments. We exploit these segments to automatically generate a large collection of cameras by casting rays orthogonal to the segments, hence towards the scene geometry since the segments represent local medial axes/mean curves. Intuitively, we are generating camera samples which follow the shape of the skeleton from far enough to provide a larger view on the overall motion of the characters. In addition, the casted rays adapt to all the geometries of the environment, including the ones not considered by the *navmesh*, creating cameras at different depths from the skeleton and through open windows/doors.

To compute these camera positions, we propose to cast rays from the skeleton in a number of heuristic directions that correspond to cinematographic camera angles *e.g.* , cameras at the same height as the characters, as well as high angle, low angle or birds'eye angles (a camera above the character). If the ray intersects the environment, we will place the camera on the ray at an given ϵ offset from the environment. If the ray does not intersect the environment, a specific threshold distance $d_{\max}$ is used to bound the position of the camera on the ray.

This heuristic sampling step is meant to be flexible and personalized by the user based on the preferred styles, by choosing the directions and the distances of the rays. The result of this step is a heterogeneous point cloud of camera locations (displayed in yellow in Fig. 5 (b)) with points resulting from a direct projection of the skeleton towards the sides, either creating an offset of the path, or adapting the offset to the scene topology (pillars, windows, etc). Each of these cameras also encompasses the direction of its associated ray and the origin of the ray on the skeleton.

4.3 Clustering

The raycast sampling step computes a cloud of camera locations from which the navigation mesh skeleton is visible. In order to compute a set of camera tracks, we propose to cluster the camera locations using a multimodel fitting algorithm use a line model. In this way we aim to identify underlying linearities both from the skeleton and the geometry of the surrounding environment. This is a problem for which many solutions have been proposed (see [Li et al. 2017] for a detailed comparison). Here we rely on a sequential

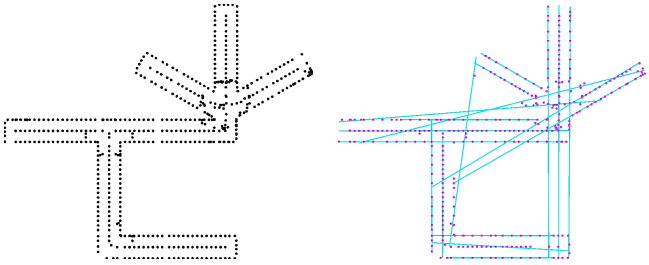


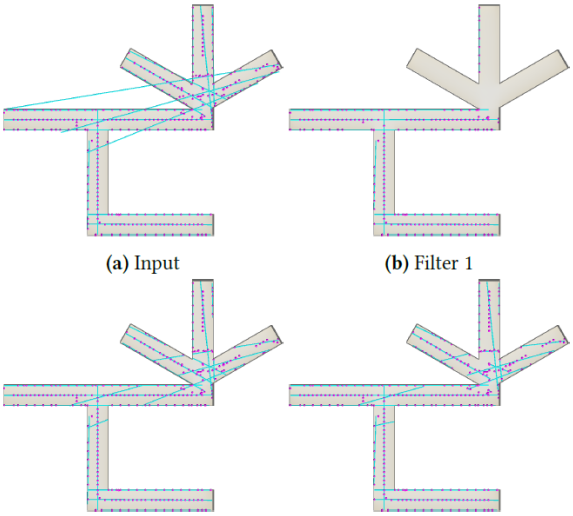
Figure 6: Clustering cameras from a cloud of camera positions, sampled from the skeleton points and raycast samples, to create linear camera tracks is performed by using a sequential RANSAC process.

RANSAC method [Fischler and Bolles 1981] which performed better than other approaches (multi-RANSAC, residual histogram analysis or J-linkage [Fouhey et al. 2010]) on our datasets, and is of $O(n)$ complexity. Given a model μ, RANSAC extracts a consensus set CS from a collection of points $\mathcal{P}$ such that:

$$CS(\mu, \mathcal{P}, \epsilon) = \{p \in \mathcal{P} | R(\mu, p) < \epsilon\}$$

where R is the error function. We used the standard point-to-line distance metric R as an error. All inliers of the first consensus set, *i.e.* $CS(\mu, \mathcal{P}, \epsilon)$, are removed from $\mathcal{P}$ and RANSAC is re-applied on the result until a given number of iterations is reached. As displayed in Figure 6 the corresponding camera tracks (in cyan) display a number of artefacts which need to be corrected through filtering.

4.4 Filtering



(a) Input **(b)** Filter 1

(c) Filters 2 and 3 **(d)** Filters 2, 3 and 4

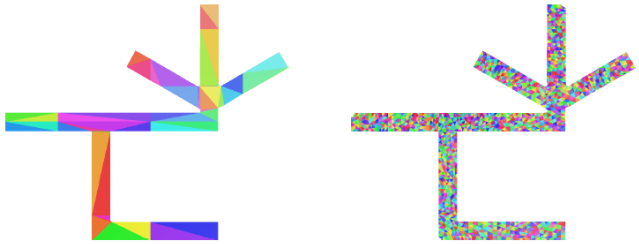
Figure 7: Computed camera tracks (in cyan) are filtered to remove artifacts which occur when clustering lines from different parts of the geometry.

The obtained camera tracks are defined by their supporting points (inliers from the sequential RANSAC) and since the clustering step only takes as input a point cloud and not the geometry

of the environment, the camera tracks might display a number of issues (*e.g.* collision with the environment) depending on the environment and the parameters. We propose four filters that the user can use in any combination, The filters are then applied sequentially, each one takes as input the output of the previous one, according to the user specification.

Filter #1 removes the parts of the tracks which collide with the environment in order to avoid the camera moving through a wall (see Figure 7 (a)). We use a iterative splitting approach to decide which segments to cut. Filter #2 removes the points have no visibility to their associated line (*i.e.* the ray between the point and its projection on the line intersects the environment). Filter #3 removes lines without supporting points, as previous filters can leave "empty" lines after a split. Filter #4 recomputes updates the equation of each line using a single RANSAC step to better fit the data after a split has occurred.

4.5 Mesh refinement and visibility estimation



(a) Original triangulation **(b)** Remeshed triangulation

Figure 8: A mesh refinement stage performed on the navigation mesh as support for visibility evaluation.

While we ensured that each camera location computed during the raycast sampling step had visibility towards a single point on the skeleton, this remains insufficient. To avoid performing visibility computation at run-time, we propose to pre-compute the camera-to-mesh visibility with the static parts of the 3D environment. We draw inspiration from [Oskam et al. 2009] which performs inter-visibility estimation for each couple of samples in the environment. To reduce the cost of the process, we only perform inter-visibility estimation from each node to each triangle of the navigation mesh, inside a limited range, using ray-casting. Prior to visibility estimation, we perform an anisotropic remeshing [Botsch and Kobbelt 2004] to refine the size all triangles of the navigation mesh (see Fig. 8(a)). This improves the precision of the visibility estimation. We store the visibility estimation both in the refined triangles of the navigation mesh (each triangle knows which cameras see it) and in the camera (each camera knows the triangles it can see). The cost in terms of memory usage grows linearly with respect to the number of cameras, and for each triangle only the degree of visibility and triangle ID is stored.

4.6 Camera navigation graph

The last step aggregates the result of the previous steps in a data structure that can be efficiently queried at run-time to compute a camera position or motion. We propose to use a non-directed graph, where each node represents a possible camera location in the environment and each edge represents possible transitions between these locations. Each node therefore needs to encode all the data necessary to efficiently place the camera: 3D position, transitions to other nodes, and the portions of environment visible from this node. The graph is computed as follows. (i) First, each camera track from the filtering step is inserted in an arbitrary order in the graph. Two nodes are created for the endpoints of a camera track, and an edge is created to link the two end points. Then,new nodes are created at the intersection between newly created edges and existing ones. This enables tracks interconnection. (ii) Second, edges are split by inserting new nodes so that each edge is shorter than a user-specified length. (iii) Third, strongly connected components in the graph are linked together by linking nodes that are closer then a user-specified threshold and ensure visibility. This enables camera tracks to easily connect to their neighbor tracks, hence creating a camera navigation graph. (iv) Lastly, points corresponding to each node are inserted into a KD-tree in order to accelerate run-time queries.

5 CAMERA PLACEMENT

In this stage we implement a simple camera placement system to illustrate the features of the camera navigation graph. We rely on the Unity's Cinemachine framing system (which smooth movements with a dampened mass-spring system) and on its virtual cameras system to reduce the impact of managing a potentially unlimited number of cameras that our system can generate (each framing a subject, that may be shared with other cameras). The inputs of our system are, for each camera, (i) a subject to frame, along with its height; (ii) a framing strategy, dictating how the subject should look on the screen; (iii) a movement strategy, dictating how the camera should move in the environment; and (iv) a cutting strategy, specifying the conditions under which a cut should be performed. To define the camera position, the system computes, at each frame, a *target* on the tracks that represents the current best possible position, given the specified strategies. Then, the actual camera position is computed by using a physical system in which a force is attracting the camera towards the target, this system, similarly to a low-pass filter, helps reducing jerkiness of the movements.

Each iteration of our algorithm, *i.e.* a frame, comprises the following steps. First, a *cutting strategy* algorithm evaluates whether a cut needs to be performed (see Section 5.1). In such case, a new target position on the tracks is computed following classical continuity rules (see Section 5.3). If no cutting is required, the target position is updated on the track using a *target moving strategy* (see Section 5.2). Once the target position is updated, the camera is moved using the force-based system. Lastly, the camera position is updated.

5.1 Cutting strategy

In order to avoid an unnecessary and expensive search for a new target position, a number of checks are performed. These checks are all the ones that do not need an updated target position, and each of them can be controlled by the user as part of the cutting strategy. There are three conditions which may trigger a cut:

- shot duration with a log normal distribution model [Galvane et al. 2015b];

- visibility check, through raycasts, to ensure the subject is not occluded for more then a given duration (200ms) by a static or a dynamic obstacle.
- framing quality, evaluating if the user-specified shot size (character on the screen) is not violated for more then a specified duration(200ms).

5.2 Target moving strategy

The target always moves on the camera tracks (the edges of the camera navigation graph). To find the optimal position for the target, we first need to select the appropriate edge, then find the right position on that edge. As the position of the target cannot be predicted too far ahead in time, the selection of the best target on the camera navigation graph is not straightforward.

We propose the following algorithm. First, a set of edges is gathered by iterating on the closest to the actor (*i.e.* edges connected to nodes that are closer than a user-specified distance). Unwanted edges (such as those not strongly connected to the previous edge) are filtered out. Then we identify potential point of interest (POI) on these edges: projection of the user on the edge line, points at the right framing distance from the actor.

Next all the POI are scored. This score is the weighted average of 4 sub-scores, with user-specified weights that constitute the framing strategy. The first sub-score is the *shot size*: using the vertical field-of-view angle of the camera and an expected on-screen height of the actor, an optimal distance camera-actor is computed. The second sub-score is the *direct visibility*, making sure that the actor is still visible by casting a ray between the POI and the actor, and monitoring if this ray intersects the environment. The third sub-score is the *indirect visibility* that tries to assess how much of the actor's surroundings are visible from the POI. This score is computed by first identifying triangles from the remeshed navmesh (see 4.5) around the actor, and computing the percentage of those that are visible from the POI using rays. The last sub-score is the *distance*, making sure that the new target position is the closest to the camera. This score is computed using the graph distance, the shortest path on the track between the two points (computed using an A* algorithm).

The new target position is then selected from the POI with the better score using a gradient descent by moving on the graph edges around the POI by fixed intervals to find the local minimal score. If, for any given reason, no point of interest can be found, a cut is needed.

5.3 Continuity rules

A cut is computed in a similar way as the target presented in 5.2 except that no edge filtering is performed. The score for each POI is computed using three of the four previous sub-scores (shot size, direct visibility and indirect visibility) and two additional cut-specific sub-scores. The first score, the *30° rule*, is derived from a classical cinematographic rule [Arijon 1991] stating that the angle between the pre-cut camera, the actor, and the post-cut camera must be over 30 degrees to avoid jump-cuts that distract the spectator. This score is computed by using the cosine of the angle between the projection of the actor's velocity vector on the pre-cut camera and its projection of the post-cut camera. The second score, the *optical flow*, derived from the "line of action" rule, saying that during a cut, the camera should not cross the line of action, so that the actor's movement, seen by the camera between the cuts, have similar directions. Computed with the cosine of the angle between the projection of the actor's velocity on the pre-cut camera and its projection on the post-cut one.

5.4 Moving the camera

Once a new target position is computed, we can move the camera towards it. We have two possibility. (i) The target is the result of a cut, then the camera can "jump" directly to the target position, and be oriented in the direction of the actor. (ii) The target is not the result of a cut, then we use a force-driven system, with two forces: one attracting the camera to the target, and a second one repulsing it from the actor. Therefore the acceleration of the camera is the weighted combination of this two forces, with a mass set to 1 to avoid "overshooting" the target and so an unpleasant back-and-forth motion of the camera. The user can also define a maximum velocity.

6 RESULTS

We show the relevance of our approach by studying a realistic scenario. All computations were done on an Intel Core i7-9850H laptop at 2.60GHz with 32GB of memory.

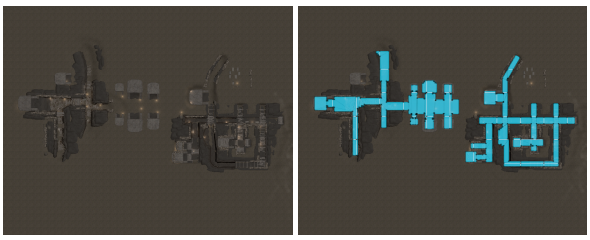


Figure 9: Trench environment (left) and correspondent navigation mesh (right) used as a benchmark scenario.

The environment elected for the scenario is a First World War-inspired trench scene available on the Unity Asset Store [1]. This environment is composed of two distinct sets of trenches (see Fig. 9) and the navigation mesh is composed of 473 triangles and 1017 vertices. Times for each pre-computation step is shown in Table 1.

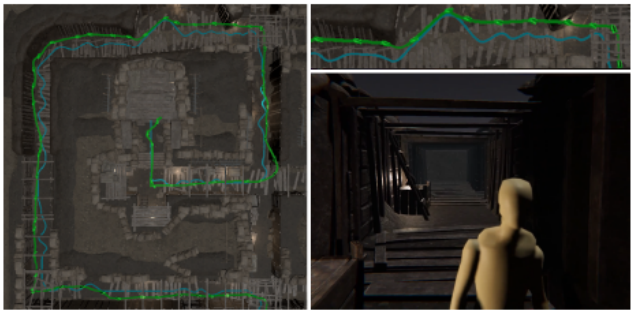
Table 1: Average time for 222 pre-computation on the environment

	Pre-computation	
	Mean	Deviation
skeleton	6.45s	2.65s
remeshing	0.04s	0.01s
raycasting	0.02s	0.01s
clustering	4.90s	10.85s
filtering	0.02s	0.01s
tracks	0.84s	0.90s

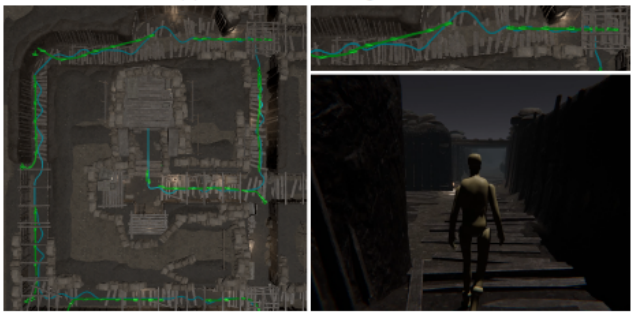
[1] https://assetstore.unity.com/packages/3d/environments/historic/world-war-trenches-152381

6.1 Artistic control

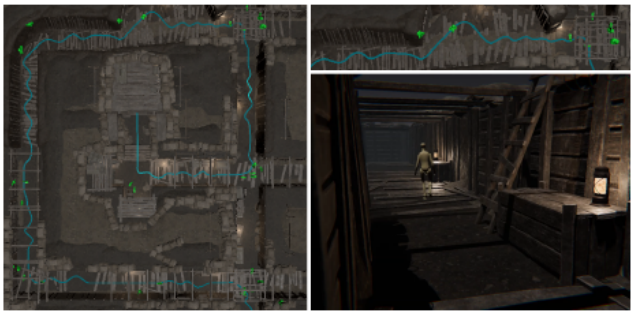
We provide a set of artistic tools to control the camera behaviours. Here we present the different styles that can be achieved using the same pre-computation step. Shown in Fig. 10 are three different camera controls in framing size (Medium shot, Long shot and Extreme long shot) and in allowed camera movement: either dynamic with no cuts, dynamic with cuts and static cameras. The trajectory of the actor (in cyan) is the same in all videos.



(a) Medium shot, cuts prohibited



(b) Long shot, cuts allowed



(c) Extreme long shot, static cameras

Figure 10: Outputs with different parameters: trajectories (on the left) of the actor (in cyan) and the camera (in green). Examples of camera frames are also provided.

Shown in Fig. 11 is the influence of the parameters during the pre-computation on the camera angles and framings that can be obtained at run-time. Kindly refer to the accompanying video for the full-length videos of all these scenarios.

(a) Raycasting only upwards

(b) Raycasting on the sides and upwards

Figure 11: Influence of the track generation on the obtained images. Actor trajectory and position are the same in both pictures.

6.2 Comparison against Probabilistic Roadmaps

We compare our graph generation approach to a probabilistic roadmap (PRM), which is a technique that generates a motion graph by randomly sampling a number of points in the environment, and linking each pair of points if the arc does not intersect the environment. We compare the two techniques on the same environment, using the same camera position algorithm described above, and having the actor take the same path.

Table 2: Cost of positioning the camera (per frame).

		Target update		Camera update	
		Mean	Dev.	Mean	Dev.
Cutting	Ours	0.01s	0.04s	0.00009s	0.00048s
	PRM	0.05s	0.02s	0.00478s	0.00444s
Not cutting	Ours	0.02s	0.03s	0.00001s	0.00001s
	PRM	0.19s	0.10s	0.00000s	0.00002s

To compare these two technique, a simple metric is to compare the time needed to compute a camera position per frame. As shown in Table 2, our method is in average 5 times quicker when looking for a new target position. This time difference is mainly due to the difference in arity between the generated graphs (*e.g.* 381 nodes and 1272 edges with our technique, 1851 nodes and 22786 edges with PRM on the same environment). This difference can be explained by the fact that while our technique tries to only generate tracks that are cinematographically interesting, PRM creates points at random, therefore needing a higher number of points for a correct result.

This effect is highlighted in Fig. 12 with a toy environment composed of a single corridor with windows. Even without taking into account the fact that our method generates tracks outside the corridor that view the inside through the windows thanks to the raycasting step, it is clear that the tracks generated via a PRM

are not as straight or useful for placing a camera. If we test PRM with a number of nodes comparable with the one generated by our techniques the PRM generates sub-graphs that do not span the entire length of the corridor, and this generates blind areas with no camera coverage, in larger and more complex environments. On the contrary, if the density of points is too high, then the entire environment is covered and a camera moving on the graph is akin to a free camera, which defeats the purpose of creating camera tracks.

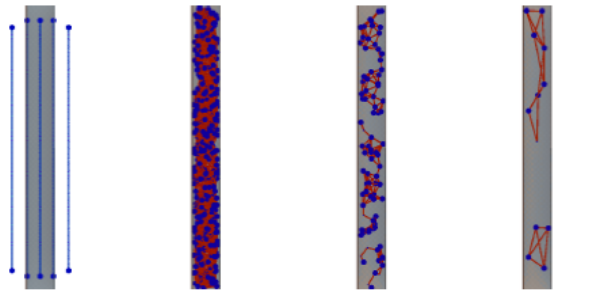


Figure 12: Visual comparison of the camera tracks. The environment is a simple corridor with windows (as in Fig. 5). In (b) point density: 1.5, link distance: 5. In (c) point density: 0.1, link distance 5. In (d) point density: 0.01, link distance 20.

7 DISCUSSION AND FUTURE WORKS

This paper addresses the problem of automatically populating 3D environments with static cameras and linear camera rails. By analyzing the navigation mesh of 3D environments, we designed different camera placement strategies which are based on an abstract representation of the environment and exploit the topology environment. We individually demonstrate the relevance of these strategies on a number of illustrative examples, and display results with all strategies on a large and complex 3D environment. By extending our approach with new camera generation strategies and a high level control, a wide range of cinematographic styles could be made available, yielding the ability for the director (should it be virtual or real) to vary in style depending on the context, events and atmosphere to convey.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 856879 (H2020 ICT-25 RIA PRESENT project).

REFERENCES

Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. 1996. A novel type of skeleton for polygons. In *J. UCS The Journal of Universal Computer Science*. Springer, 752–761.

Daniel Arijon. 1991. *Grammar of the film language.* Silman-James Press.

Mario Botsch and Leif Kobbelt. 2004. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 185–192.

Jonathan W. Brandt and V.Ralph Algazi. 1992. Continuous skeleton computation by Voronoi diagram. *CVGIP: Image Understanding* 55, 3 (5 1992), 329–338.

Luca Chittaro, Lucio Ieronutti, Roberto Ranon, Eliana Siotto, and Domenico Visintini. 2010. A high-level tool for curators of 3d virtual visits and its application to a virtual exhibition of renaissance frescoes. In *Proceedings of VAST*, Vol. 2010. 11th.

Marc Christie and Patrick Olivier. 2006. Camera Control in Computer Graphics. *EUROGRAPHICS* (2006).

Marc Christie, Patrick Olivier, and Jean-Marie Normand. 2008. Camera control in computer graphics. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 2197–2218.

Martin A. Fischler and Robert C. Bolles. 1981. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Graphics and Image Processing* 10064 (March 1981).

David F Fouhey, Daniel Scharstein, and Amy J Briggs. 2010. Multiple plane detection in image pairs using j-linkage. In *2010 20th International Conference on Pattern Recognition*. IEEE, 336–339.

Quentin Galvane, Marc Christie, Chrsitophe Lino, and Rémi Ronfard. 2015a. Camera-on-rails: automated computation of constrained camera paths. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*. ACM, 151–157.

Quentin Galvane, Christophe Lino, Marc Christie, Julien Fleureau, Fabien Servant, Fran¸ois-louis Tariolle, and Philippe Guillotel. 2018. Directing cinematographic drones. *ACM Transactions on Graphics (TOG)* 37, 3 (2018), 1–18.

Quentin Galvane, Rémi Ronfard, Christophe Lino, and Marc Christie. 2015b. Continuity editing for 3D animation. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Nicolas Halper, Ralf Helbing, and Thomas Strothotte. 2001. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. In *Computer Graphics Forum*, Vol. 20. Wiley Online Library, 174–183.

Hui Huang, Dani Lischinski, Zhuming Hao, Minglun Gong, Marc Christie, and Daniel Cohen-Or. 2016. Trip Synopsis: 60km in 60sec. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 107–116.

Hongda Jiang, Bin Wang, Xi Wang, Marc Christie, and Baoquan Chen. 2020. Example-driven Virtual Cinematography by Learning Camera Behaviors. *ACM Transactions on Graphics (TOG)* 39, 3 (2020).

Fabrice Lamarche. 2009. Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 649–658.

Jing Li, Tao Yang, and Jingyi Yu. 2017. Random sampling and model competition for guaranteed multiple consensus sets estimation. *International Journal of Advanced Robotic Systems* 14, 1 (2017).

Tsai-Yen Li and Chung-Chiang Cheng. 2008. Real-time camera planning for navigation in virtual environments. In *International Symposium on Smart Graphics*. Springer, 118–129.

Christophe Lino, Marc Christie, Fabrice Lamarche, Guy Schofield, and Patrick Olivier. 2010. A real-time cinematography system for interactive 3d environments. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 139–148.

Amaury Louarn, Marc Christie, and Fabrice Lamarche. 2018. Automated staging for virtual cinematography. In *Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games*. ACM, 4.

Dennis Nieuwenhuisen and Mark H Overmars. 2004. Motion planning for camera movements. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, Vol. 4. IEEE, 3870–3876.

Ramon Oliva and Nuria Pelechano. 2011. Automatic generation of suboptimal navmeshes. In *International Conference on Motion in Games*. Springer, 328–339.

Joseph O'Rourke. 1987. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., New York, NY, USA.

Thomas Oskam, Robert W Sumner, Nils Thuerey, and Markus Gross. 2009. Visibility transition planning for dynamic camera control. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 55–65.

Andrea Tagliasacchi, Ibraheem Alhashim, Matt Olson, and Hao Zhang. 2012. Mean curvature skeletons. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1735–1744.

Anton van den Hengel, Rhys Hill, Ben Ward, Alex Cichowski, Henry Detmold, Chris Madden, Anthony Dick, and John Bastian. 2009. Automatic camera placement for large scale surveillance networks. In *2009 Workshop on Applications of Computer Vision (WACV)*. IEEE, 1–6.

Kun Zo Xiang Xu, Min Huang. 2011. Automatic Generated Navigation Mesh Algorithm on 3D Game Scene. *Procedia Engineering* 15 (2011), 3215–3219.